Machine-independent virtual memory

1. Big ideas
    a. Abstract low-level details of privileged architecture in implementation
        i. More variation in virtual memory than CPU scheduling
        ii. Closer ties to whole OS than devices (via device drivers)
    b. Solution: clever data structures
        i. Make copy-on-write efficient
        ii. Treat hardware structures as a cache of real data – "soft state" can be discarded & regenerated
2. Goals
    a. Multiprocessor
    b. Unix compatibility
        i. One of first OS designed to be compatible with another one!
    c. Message based (not procedure oriented)
    d. Multiprocessor – capable
    e. Network – capable
3. Mach overview
    a. Mach abstractions
        i. Task = execution environment / address space / unit of resource allocation
        ii. Thread = unit of CPU utilization
        iii. Port == communication channel, a queue for messages protected by capabilities
            1. Are basically capabilities you invoke by sending a message instead of dereferencing
            2. Q: How do you get a port? From a name server or from your parent
            3. NOTE: like Mach
        iv. Message = typed collection of data, may contain ports
        v. Memory object = collection of data provided for and managed by a server that can be mapped into an address space
    b. Operations on objects
        i. For everything but messages, implemented by sending/receiving messages
        ii. Indirection of messages allows a network to be interposed, either an SMP or a cluster / distributed system
        iii. Integrated VM and IPC reduces performance overhead of IPC compared to shared memory ; need not copy
4. Memory goals
    a. Flexible use of VM capabilities

        i. Software shared memory: multiprocessor over the network
        ii. Use of pages for guard pages (unmapped) for garbage collection or allocation
        iii. Could do compression instead of swapping easily
    b. Machine independent:
        i. Different CPUs have different VM design:
            1. X86: hardware radix tree read by CPU
            2. MIPS: software filled TLB
            3. RT (power): inverted page table (big hash table) read by CPU
            4. SPARC: few Segments, TSB, software TLB
            5. VAX: virtual page table: linear mapping of pages in virtual address space, can selectively map to physical pages to get sparseness
        ii. How handle portability in an OS?
            1. Linux / Unix / Windows approach: pick one architecture as the OS implementation, emulate on others
                a. Linux, Windows: x86
                b. Unix: Vax
            2. Why is this bad?
                a. Don't get to use the features of other systems
                    i. E.g. multiple page sizes
                b. Extra overhead for multiple data structures after emulation
                    i. Put mappings in two places
            3. Goal: provide an abstraction not of virtual memory, but of virtual memory hardware
                a. What is it: a software TLB (the pmap)
5. Implementation of mach memory
    a. Separate into two problems:
        i. Machine dependent: what the HW requires
            1. When used?
                a. Only for operations the HW must know about: manipulating translations within an address space
        ii. Machine independent: efficient structures for OS-level operations, not tied to HW
            1. Memory-mapped files
            2. Copy-on-write
            3. User-level paging (more on this)
            4. Fine-grained protection changes at user level
        iii. A bit like ExoKernel, but some abstraction

6. Machine-independent data structures
    a. Abstract address space:
        i. address maps: sorted linked lists of map entries, each describing a region, per task: protection + inheritance.
        ii. Used for PF lookups, copy/prot operations, allocation/deallocation of address ranges
    b. Memory objects:
        i. units of backing storage:
    c. Memory regions (stack, heap, memory-mapped file)
        i. specifies resident pages (those in DRAM) + where to find non-resident pages.
        ii. Non-resident pages can be stored outside kernel
    d. Copy-on-write:
        i. Shadow objects shadow a memory object and contain COW pages
        ii. Show example:
            1. Have base memory object
            2. When CoW, create shadow object that points at base object
                a. Addres Map points at shadow object
                b. Shadow object only has pages not in base object
            3. On CoW, allocate new shadow object, points to next shadow object
        iii. share maps for explicitly shared memory (not COW) == layer of indirection for an address map
            1. Address map points at share map
            2. Share map points to underlying memory objects
            3. Adds an indirection as don't have to manipulate underlying objects, e.g. when forking() and coping the whole address space
    e. Physical memory:
        i. Treated as a cache of parts of memory objects
        ii. resident page table: current attribute for all physical pages
        iii. Keeps track of how pager is being used: as part of an object
        iv. Also indexed by offset into object for page faults / tlb misses
7. Machine-independent structures
    a. pmaps: subset of pages visible to HW -
    b. Is a **coherent cache** of machine-independent state.
    c. can be thrown away any time for efficiency or space; can be reconstructed.
    d. QUESTION: Why?

        i. Can save memory by not maintaining
        ii. Can make operations more efficient by not keeping up-to-date; just delete it

8. What happens on a page fault / TLB miss?
    a. First consult pmap to see if there is already a mapping. If so, use it
    b. If not, call machine-independent code to look at address map, to find the appropriate memory object, then in the object/offset hash table to find corresponding physical page
    c. If not in memory, deal with paging (coming up soon)
9. Operations on objects
    a. Allocate / deallocate
    b. Set protection / inheritance status
        i. Set on memory regions, propagate to pmaps
    c. Create & manage a memory object for other tasks
    d. Optimizations
        i. Read/write sharing and COW sharing
        ii. Whole address space can be sent with no copying!
            1. E.g. used for Unix FORK
            2. Implemented with shadow map that specifies real map to receive page from on fault
        iii. Protection
            1. Can set current protection – in use by hardware, and maximum – limit to which it can be lowered (e.g. prevent making it writable)
            2. BIG IDEA: like mprotect(); allow programs to use hardware features if not needed for protection/security
10. Memory / communication
    a. Goal: make communication fast by using memory
    b. QUESTION: How?
        i. Make it easy to send large-objects
        ii. Only copy data when necessary; otherwise re-use same data via sharing
        iii. Allow external sources to manage data
        iv. QUESTION: How easy is this to use? A: have to pack data onto a single page; still have to marshall/unmarshall. Mapping address may be different in different address spaces.
    c. High-level structure
        i. AS contains memory regions (ranges of addresses that are mapped to something)
        ii. Mach flexible controls what they are mapped to for efficient read/cow/rw sharing
        iii. External pagers for backing pages

1. Memory object represents a data object obtained from an external pager

11. External pagers
    a. What are they?
        i. BIG Abstraction:
            1. Kernel maintains in-memory cache of an object
            2. Kernel invokes pager when it moves things in/out of cache
            3. Pager invokes kernel when things are unavailable
        ii. Kernel paging daemon handles physical pages
            1. Looking for pages to replace (e.g. clock, LRU)
            2. Tracking free pages
            3. Caching common memory objects (e.g., common executable code)
        iii. COMMENT: Think caching
            1. Kernel is simple cache for data
            2. Complexity handled by pager
                a. Moving data in/out of cache (abstracts path to backing storage)
                b. Indicating things should stay in cache longer or should be removed sooner
        iv. COMMENT: think layer of indirection
            1. Kernel provides layer between program & pager
            2. Kernel makes pager data available in address space
        v. Provides initial data for memory object
        vi. Controls access to memory object (e.g. when can you r/w)
        vii. Provides backing for memory object (e.g. when it is evicted)
        viii. Interface:

- vm_allocate_with_pager creates one in task at an address. Called by
an application, memory object specifies the pager

- kernel to dm interface: (async)
- init - init a mem objc
- data_request - request data be filled in
- data_write - write back data
- data_unlock - unlock data - on a permission fault
- data_create -

- dm callbacks to kernel:
- data_provided; supply memory contents
- lock: restricts access to a page - e.g. read onl
- flush: invalidates cache, may writeback, kick from cache

- clean_request: force data writeback, but can keep in cache
- cache: kernel should keep objects around if not in use (e.g. program will be run again soon).
- data_unavailable: notify that no mem available

Note: decoupling of data_request and data_provided; can return more data than requested (e.g. prefetching)

b. Benefits:
   i. Most of kernel memory is treated as a cache – transparent mixing of file cache with VM system allow a larger file cache (Unix used just 10% at the time)
   ii. Fast access to large shared objects – e.g. shared array access

c. How are they used?
   i. File system with whole-file access
      1. Model: file system server process + FS DM
      2. File APIs RPC to FS server
         a. open file: RPC to FS server to create memory region, returns a COW of the region
      3. Memory access to file
         a. page fault causes pagee_data_request to FS DM
         b. FS DM calls disk to get data, provides data to kernel for
         c. Kernel creates COW for client of the page
      4. When closed, can flush back to disk (not shown in example)
   ii. Consistent shared memory
      1. Idea: allow processes on different systems to share memory
      2. Approach:
         a. Have a server responsible for a page
         b. Ask that server for the page
         c. It provdes it to as many readers as want it
         d. When get a call to change protection (pager_data_unlock), flushes page from other systems, THEN updates local protection
   iii. Process migration
      1. Can move processes to other systems for load balancing
      2. Use consistent shared memory to fault pages over as accessed
   iv. Transactions
      1. Can allow DB to have control over paging of data

        2. Can provide transactional memory; by logging writes before updating structures on disk
    v. Idea: easy to implement things like this
  d. Problems:
    i. What if pager doesn't respond?
        1. A: have default pager that flushes pages to disk
        2. Kernel knows about default pager, calls it when other pager fails. Are not multiple default pagers.
    ii. TRUST: must a process trust its pager?
        1. It has access to all the data
        2. What if share with a more trusted process (e.g. OS process vs application) from an untrusted pager?
  e. QUESTION: What is the cost?
    i. Overhead of calling to usermode
    ii. Trusted third parties
  f. Big picture
    i. Allow memory to be used for communication, not just local storage
    ii. Provide interface for external pagers to get involved on important decisions; where data comes from, invalidating data
    iii. Efficient communication by sharing memory
    iv. Treat kernel as a cache for data from other places; like kernel-managers in pilot

12. ISSUES:
  a. Was Mach successful? Pretty much the only research OS to see commercial use
  b. Supporting multiple OS never worked well; too hard to be compatible with MS OS
  c. Cost of IPC too high; unix server moved into kernel
  d. MacOS
    i. Mach for IPC, process & thread management, memory management, hardware abstraction